UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Conversor de archivos generados por cámaras de eventos

Autor: Andrés Calderón Ayuso
Tutor: Erik Velasco Salido
Ponente: José María Martínez Sánchez

junio 2021

Andrés Calderón Ayuso
Conversor de archivos generados por cámaras de eventos

Andrés Calderón Ayuso
C\ Erasmo de Rotterdam Nº 5 A311

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Agradecimientos

En primer lugar, quiero agradecer a Erik y a José María por permitirme escoger este proyecto. En especial agradecerles el constante apoyo dado y la impecable orientación recibida durante todos estos meses de trabajo.

Agradecer a mis compañeros de la universidad y amigos, por todos los buenos momentos vividos y por haber estado ahí siempre que surgía cualquier dificultad. Todos ellos me han acompañado en este maravilloso viaje de cuatro años, dándome apoyo y ánimos desde el primer hasta el último momento.

En especial quiero agradecer a mi pareja, por haberme ayudado y animado en los peores momentos y por ser la luz que ilumina mis días.

Y por encima de todo, agradecer a mi familia por haberme apoyado desde siempre. Por haber creído en mí en los momentos más difíciles y por nunca darme la espalda. Muchas gracias por haberme alegrado en los días más oscuros y por guiarme en la vida para poder crecer como una gran persona. Gracias del fondo de mi corazón.

Resumen

Con el crecimiento de las tecnologías y de los centros de investigación de estos últimos años, se ha desarrollado un nuevo tipo de cámara que no necesita obturador, las cámaras de eventos.

Estas cámaras capturan asíncronamente el cambio de brillo por cada píxel que recojen, de tal manera que generan multidud de eventos, cada uno con sus coordenadas, el momento del tiempo en el que han sucedido y si ha habido incremento o decremento de brillo.

Diversas universidades y centros de investigación se han subido al carro de esta nueva tecnología y han empezado a desarrollar sus propias cámaras de eventos junto a multitud de simuladores y herramientas para su análisis.

Debido a esta aparicion de diversos grupos de investigación antes de que existiese un consenso global sobre la estructura de los archivos generados por las cámaras, se han llegado a definir diferentes formatos de archivos para guardar los eventos producidos por estas cámaras.

El objetivo de este proyecto es desarrollar un conversor de archivos generados por cámaras de eventos, de tal manera que cualquier investigador que necesite cambiar de formato de archivo lo pueda realizar fácilmente con este programa.

La aplicación desarrollada está subida en un repositorio de GitHub [1] junto al README preparado para guiar al usuario.

Palabras clave

Conversor de archivos, conversor de eventos, cámaras de eventos, evento, rosbag, aedat, matlab, bin, txt, hdf5

Abstract

With the growth of technologies and research centers in recent years, a new type of camera has been developed that does not need a shutter, the event cameras.

These cameras asynchronously capture the change in brightness for each pixel they collect, in such a way that they generate a multitude of events, each with its coordinates, the moment in time in which they have happened and if there has been an increase or decrease in brightness.

Some universities and research centers have jumped on the bandwagon of this new technology and have begun to develop their own event cameras together with a multitude of simulators and tools for their analysis.

Due to the appearance of various research groups before there was a global consensus on the structure of the files generated by the cameras, different file formats have been defined to save the events produced by these cameras.

The objective of this project is to develop a converter for files generated by event cameras, in such a way that any researcher who needs to change the file format can easily do it with this program.

The developed application is uploaded in a GitHub repository [1] together with the README prepared to guide the user.

Keywords

File converter, event converter, event cameras, event, rosbag, aedat, matlab, bin, txt, hdf5

Índice

Listas

Lista de algoritmos

Lista de códigos

Lista de cuadros

Lista de ecuaciones

Lista de figuras

Lista de tablas

Lista de cuadros

1

Introducción

En esta primera sección se recogen las principales motivaciones del proyecto y los objetivos que han guiado el desarrollo del programa. Posteriormente, se explicará la organización del documento.

1.1. Motivación

El mundo de las cámaras de eventos lleva expandiéndose estos últimos años entre diversos grupos de investigación de todo el mundo [2, 3]. Estas cámaras funcionan de manera totalmente diferente a las usuales, se centran en capturar cambios de brillo en lugar de imágenes. A cada cambio de iluminación recogido en un momento determinado se le llama evento. De esta forma, estas cámaras acaban generando listas de eventos recogidos durante su grabación.

Las cámaras de eventos se fueron desarrollando sin que los grupos de investigación propusieran una configuración común para describir los eventos captados por las cámaras. Esto conllevó a la aparición de un conglomerado de formatos, la mayoría incompatibles entre sí.

Sin embargo, todas estas configuraciones de archivo siempre almacenan unos datos en común:

- Cada archivo contiene una lista de los eventos captados por la cámara durante su funcionamiento.
- Cada evento se compone siempre de las mismas 4 variables:
  - Su coordenada x.
  - Su coordenada y.
  - El momento del tiempo en el que ha sucedido el evento.
  - La polaridad del evento.

Además de estos datos, algunos formatos contienen diversa información adiccional a los eventos, como puede ser la calibración de la cámara [4] o la información generada por el *Inertial Measurement Unit* (IMU) [5]. Esta misma información en otros tipos de archivos es opcional o incompatible.

El principal problema a la hora de guardar los datos en un archivo, es que cada uno utiliza un formato diferente para el almacenamiento. Por ejemplo, los archivos utilizados por el simulador ESIM [6] utilizan el formato *.bag*, mencionado en el siguiente artículo [7]. Sin embargo, el clasificador NVS2Graph [8]

utiliza como datos de entrada archivos de tipo *.mat*, usados por la aplicación *Matrix Laboratory* (MATLAB) , cuyo formato se explica en la sección 3.4.

Esta disparidad de formatos no ha sido resuelta todavía y a día de hoy se siguen desarrollando aplicaciones, simuladores y cámaras de eventos que almacenan los datos con diferentes configuraciones de archivo. Para paliar este problema, se ha propuesto desarrollar una aplicación que sea capaz de cambiar el formato de un archivo entre una serie de formatos admitidos. De esta manera, cualquier investigador podría usar programas que antes eran incompatibles entre sí utilizando el programa implementado en este proyecto.

1.2. Objetivos

Tras aclarar las motivaciones de este proyecto, se especifican los objetivos seguidos a la hora de desarrollar el programa.

O-1.– Investigar sobre las cámaras de eventos y su estado actual.

O-2.– Analizar los diferentes formatos de archivos que las cámaras de eventos utilizan.

- **O-2.1.–** Escoger los formatos más usados para implementar su conversión.
- **O-2.2.–** Investigar acerca del formato propio de cada uno y de las posibles versiones que existan dentro del mismo.

O-3.– Especificar un formato común al que transformar el resto de los formatos.

O-4.– Desarrollar el código correspondiente a los conversores, un programa por cada tipo de formato.

- **O-4.1.–** Desarrollar el programa que lea cualquier archivo y lo transforme al formato común.
- **O-4.2.–** Desarrollar el programa que, a partir del formato común, sea capaz de escribir cualquier archivo.

O-5.– Desarrollar varias Interfaz de Usuario (IU) para facilitar el uso del programa al usuario.

- **O-5.1.–** Desarrollar una interfaz gráfica.
- **O-5.2.–** Desarrollar una interfaz basada en la terminal.
- **O-5.3.–** Desarrollar una interfaz que no muestre nada, solo los errores.

O-6.– Crear un archivo de configuración para automatizar las conversiones.

O-7.– Programar el analizador de argumentos del programa.

O-8.– Desarrollar el programa principal juntando todos los componentes.

O-9.– Redactar la documentación necesaria para el uso del programa.

- **O-9.1.–** Redactar el manual del usuario.
- **O-9.2.–** Redactar un documento especificando las diferentes configuraciones de archivo admitidas y su formato.

O-10.– Crear varios scripts de test para comprobar el correcto funcionamiento del programa.

O-11.– Subir el programa a la plataforma GitHub.

O-12.– Proponer posibles mejoras de cara al futuro.

1.3. Organización del documento

En esta sección se mostrará la organización del documento. Este está dividido en los siguientes capítulos:

En el **primer capítulo** se ha presentado el proyecto, se han definido las motivaciones y los objetivos y se ha explicado la estructura del documento.

En el **segundo capítulo** se presenta el estado del arte haciendo una breve introducción a las cámaras de eventos. También se han explicado los formatos usados actualmente y las tecnologías empleadas en el proyecto.

En el **tercer capítulo** se muestran los diferentes formatos existentes de archivos generados por cámaras de eventos. También se expone las implementaciones de escritura y lectura para cada formato.

En el **cuarto capítulo** se expone el desarrollo del sistema, explicando el sistema modular desarrollado y el uso e implementación de cada módulo por separado. Además, se analiza el contenido del programa principal.

En el **quinto capítulo** se muestran las diferentes pruebas realizadas para garantizar el correcto funcionamiento de la aplicación.

En el **sexto capítulo** se exponen las conclusiones obtenidas tras el desarrollo del programa y posibles mejoras de cara al futuro.

Al final del documento se muestra el apéndice. Este incluye el contenido del archivo **README** asociado al repositorio del proyecto desarrollado.

2

Estado del arte

Este capítulo se centrará en el estudio del estado del arte de los distintos formatos con los que las cámaras de eventos funcionan. Además se añadirá una breve introducción a las cámaras de eventos y se analizarán diversos conversores usados hoy en día.

2.1. Cámaras de eventos

Para comenzar el capítulo, se procederá a mostrar una breve introducción al mundo de las cámaras de eventos.

Estas cámaras no generan los fotogramas estándar a los que estamos acostumbrados. Son sensores de visión bioinspirados que capturan cambios de brillo en un píxel determinado [9]. Estos cambios de brillo generan lo que se conoce como **evento** y son capturados asíncronamente. Un evento se compone de cuatro variables:

1.– La coordenada en el eje X donde ha sucedido el cambio de brillo.

2.– La coordenada en el eje Y donde ha sucedido el cambio de brillo.

3.– El instante de tiempo en el que ha sucedido el cambio de brillo, también conocido como *timestamp*.

4.– La polarización del evento, indicando si ha habido incremento o disminución de brillo.

Las cámaras de eventos emiten en su funcionamiento un listado de los eventos sucedidos en su campo de visión. Una vez acabada la grabación, estos datos se pueden procesar para diversas utilidades, por ejemplo, para la clasificación de objetos [8].

Como la salida de estas cámaras se compone de una secuencia de eventos asíncronos en lugar de las convencionales imágenes, los algoritmos tradicionales de análisis de imágenes no se pueden aplicar en estas cámaras. Por este motivo se requieren nuevos algoritmos que exploten la alta resolución temporal y la naturaleza asincrónica del sensor [9].

A día de hoy, muy pocas empresas trabajan con las cámaras de eventos. Las empresas que los utilizan tienen estos artefactos en la sección de investigación, a la espera de que salgan nuevas aplicaciones en este campo. Las principales empresas que producen cámaras de eventos son: **iniVation**,

CelePixel y **Prophesee**. Solo dos grandes empresas han intentado trabajar con estas cámaras, **Samsung** y **Sony**. De estas dos empresas, solo **Samsung** las ha intentado comercializar masivamente [10].

Existen multitud de tipos de cámaras de eventos. Caben destacar dos, el *Dynamic Vision Sensor* (DVS) y el *Dynamic and Active-pixel Vision Sensor* (DAVIS) .

DVS: Esta cámara produce eventos con polaridad e intensidad asociada. En este vídeo se puede observar su funcionamiento [11].

DAVIS: Esta cámara funciona exactamente igual que el DVS pero se añade un *Active Pixel Sensor* (APS) para producir cuadros de imágenes junto a los eventos. En este vídeo se puede observar su funcionamiento [12].

Muchas cámaras de eventos tienen además una IMU acoplada, que nos aporta información sobre la velocidad, la orientación y las fuerzas gravitacionales que recibe la cámara mientras está en funcionamiento [5].

En la figura 2.1 se puede observar un ejemplo de una representación gráfica de los eventos generados por una cámara de eventos. En esta imagen, cada punto azul representa un evento. Estos van apareciendo a lo largo del tiempo de grabación a medida que van produciéndose cambios de brillo en la entrada. Además, cada evento sucede en un píxel determinado, representado por su coordenada en el eje x y su coordenada en el eje y. En esta gráfica no se representa la polarización de los eventos. Fuente [7].

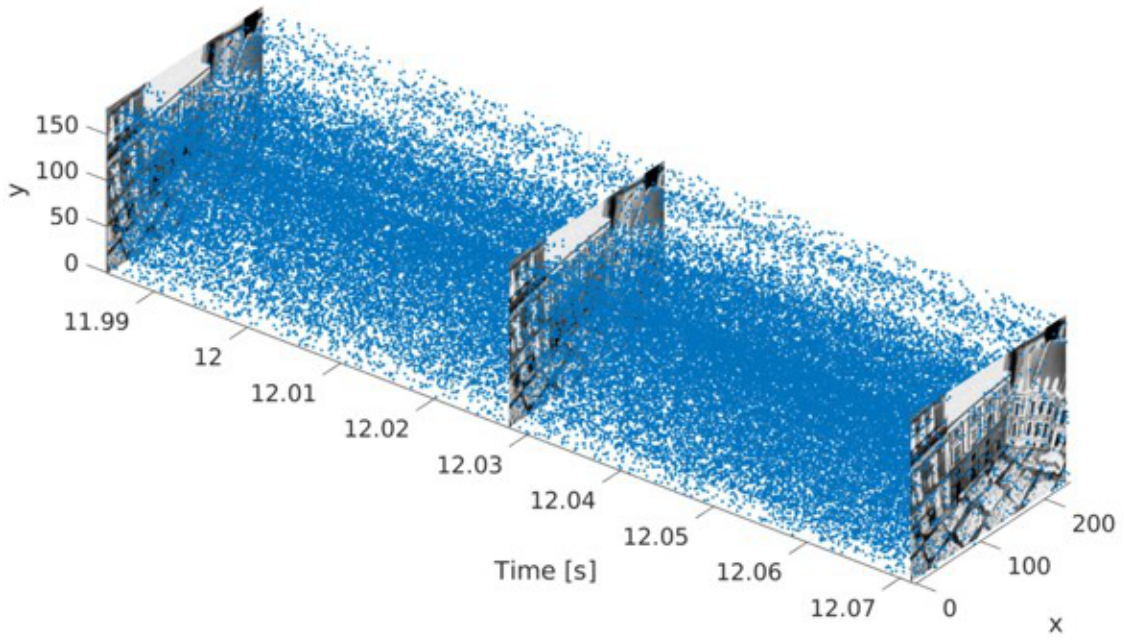

Figura 2.1: Gráfico que representa los eventos grabados por una cámara de eventos.

2.2. Principales formatos

En el mundo de las cámaras de eventos existen diversos formatos para almacenar la lista de eventos generada. Antes de empezar a desarrollar el proyecto, se han analizado los diferentes formatos existentes y sus aplicaciones de escritura o lectura. Para ello, se analizaron los diferentes *datasets* existentes y sus formatos. Alguno de los *datasets* utilizados en este análisis son: [7, 8, 13–19]. Se puede observar que alguno de estos *datasets* contienen los datos en varios formatos. Esto ha ayudado a la realización de las pruebas del sistema, para más información mirar capítulo 5.

Tras esta investigación, se han encontrado un total de cinco formatos diferentes:

bag: Formato que utilizan los archivos utilizados por *Robot Operating System* (ROS) 3.2.

mat: Formato utilizado por la aplicación MATLAB 3.4.

txt: Archivos de texto plano 3.3.

bin: Archivos binarios 3.5.

aedat: Creado específicamente para almacenar eventos 3.6.

hdf5: Diseñado para almacenar grandes cantidades de datos.

Se han encontrado diversas aplicaciones útiles que han ayudado al desarrollo del código. Todas estas aplicaciones vienen referenciadas en la sección *Utilities* del siguiente artículo [20].

2.3. Tecnologías empleadas

Antes de comenzar el desarrollo del programa, se realizó un análisis de las tecnologías que se utilizarían.

Se comenzó decidiendo el lenguage en el que se iba a implementar todo el código. El lenguaje escogido fue **Python** debido a que la mayoría de librerías de lectura y escritura de los formatos a implementar estaban desarrolladas en Python. Además, es uno de los lenguages más utilizados hoy en día. Cientos de grandes empresas como **Google** o **Dropbox** lo utilizan. Esto da lugar a una comunidad muy activa, que cuida el entorno así como sus actualizaciones [21].

Tras la aparición de la idea del archivo de configuración, apartado 4.4.2, se decidió guardar estos datos en formato *JavaScript Object Notation* (JSON) , ya que es un lenguaje fácil de leer y escribir tanto para humanos como para ordenadores [22].

Después de escoger los lenguages a utilizar, se manejo la posibilidad de usar un *Integrated Development Environment* (IDE) . Estas aplicaciones ofrecen servicios que facilitan el desarrollo de software al programador. Se optó por usar el IDE **PyCharm** de la empresa **JetBrains** [23]. Este entorno está diseñado para ayudar en la programación de proyectos en Python. Ofrece algunos servicios muy útiles

como puede ser el depurador o el verificador de código **Pylint** [24].

Por último, se utilizaron aplicaciones para visualizar el contenido de los archivos en algunos formatos. Para verificar el contenido de los mensajes de tipo *bag*, se utilizó la versión Noetic de ROS , la cual nos permite leer los mensajes a través de la terminal [25]. En el caso de los archivos *mat*, se utilizó la aplicación MATLAB . Esta aplicación puede mostrar el contenido de cualquier archivo de tipo *mat* [26]. Para los archivos *txt*, se utilizó PyCharm. El resto de archivos no tienen aplicaciones de lectura o escritura conocida.

3

Formatos

En este capítulo se recogen todos los formatos admitidos por el programa desarrollado. Además, se aclaran las librerías utilizadas para poder usar estos formatos. Al principio del capítulo se explica el formato común seguido, el porqué de su estructura y su implementación.

Destacar antes de empezar que el formato *Hierarchical Data Format 5* (HDF5) no se ha podido implementar. Esto se debe a que los eventos guardados en estos ficheros no siguen las misma estructura que en el resto de formatos. Además, apenas existen *datasets* que utilicen este formato, con lo que se ha optado por no implementarlo en el programa.

3.1. Formato común

Para empezar el capítulo, se explicará el formato común elegido que sirve como nexo de unión entre el resto de formatos.

Este formato debe admitir todas las variables que un evento contiene además de especificar el tipo y magnitud de cada una.

También debe tener en cuenta el lenguage elegido para este proyecto, el cual fue Python por la gran cantidad de librerías que tiene para leer, escribir y manipular datos en multitud de diferentes configuraciones de archivo.

Tras realizar el análisis de los formatos escogidos, se implementó un formato común a todos ellos. En la tabla 3.1 se puede observar el nombre y tipo de cada variable que contiene este formato.

Nombre de la variable	Tipo de dato	Variable que representa
x	integer	coordenada x
y	integer	coordenada y
ts	float	*timestamp*
pol	bool	polarización

Tabla 3.1: En esta tabla se especifica el formato común utilizado para la conversión de los archivos.

Destacar que el *timestamp* de este formato está expresado en segundos. En algunos formatos esta variable se guarda con diferente magnitud, cosa que hay que tener en cuenta a la hora de realizar la conversión.

También añadir que la variable polarización cuando es verdadera indica aumento de brillo, y cuando es falsa indica disminución de brillo.

3.1.1. Implementación del formato común

Este formato fue implementado mediante una clase en Python de nombre *Event*. El contenido de esta clase se muestra en el apartado 3.1.1. Esta clase solo implementa el constructor, puesto que solo sirve para almacenar datos, no necesita ningún método de clase.

Se puede observar que cada variable es casteada al tipo correspondiente a la hora de instanciar un objeto de esa clase. De esta manera, si un parámetro no es del tipo adecuado, nos salta la excepción correspondiente para luego poder capturarla e informar al usuario del error.

Código 3.1: Se presenta el código correspondiente a la clase Event, la cual al instanciarse representa un evento producido por una cámara de eventos.

```
1 class Event:
2     def __init__(self, x, y, pol, ts):
3         self.x = int(x)
4         self.y = int(y)
5         self.pol = bool(pol)
6         self.ts = float(ts)
```

Para almacenar todos los eventos de un fichero, se crea una lista de Python vacía y se van añadiendo las instancias de los eventos correspondientes.

Después, para escribirlos en un nuevo fichero, se itera sobre la lista de eventos y se van escribiendo en el archivo correspondiente con su formato adecuado.

3.2. Archivos bag

El primero de los formatos a explicar es el formato que utiliza ROS , ya que es el más utilizado en el campo de las cámaras de eventos. Estos archivos se utilizan en multitud de disciplinas y se denominan **bags**. Además, todos deben seguir el formato especificado en su web oficial [27]. En este caso la versión que implementamos fue la v2.0, ya que es la utilizada actualmente. La extensión que utilizan estos archivos es *.bag*.

Los archivos bag guardan un conjunto de mensajes de tipos diversos, cada uno de estos mensajes

viene definido por un archivo *.msg* que indica las variables que contiene. Cada mensaje contiene un *header* propio con información relevante para ROS . Además, cada uno de los mensajes guardados se ve vinculado obligatoriamente con uno de los tópicos que el bag contiene.

En el caso de los archivos bag usados para cámaras de eventos, todos los eventos se ven envueltos bajo un tópico común que varía según decida el creador del archivo. Por ejemplo, es muy común, en las cámaras DVS usar el tópico */dvs/events*.

En estos archivos los eventos se pueden guardar en mensajes de dos tipos diferentes:

Event: Este tipo de mensajes guarda un único evento por mensaje, cuyas variables vienen especificadas en el siguiente archivo [28].

EventArray: Este tipo de mensaje guarda un array de eventos, además de la altura y anchura de la matriz de píxeles que graba la cámara. Su formato viene especificado en el siguiente enlace [29].

Como se puede intuir, el tipo *EventArray* crea un array de estructuras de tipo *Event*, por lo tanto nos centraremos en analizar el formato de este último.

El tipo *Event* contiene cuatro variables, en la tabla 3.2 se presentan las variables, los tipos y la variable común que representa cada parámetro del evento.

Nombre de la variable	Tipo de dato	Variable que representa
x	unsigned integer de 16 bits	coordenada x
y	unsigned integer de 16 bits	coordenada y
ts	estructura time propia de rosbag	*timestamp*
polarity	booleano	polarización

Tabla 3.2: En esta tabla se especifica el formato de los eventos contenidos en los archivos bag.

La estructura time de rosbag contiene los segundos y los nanosegundos en dos variables separadas. Para obtener el *timestamp* del evento se deben juntar ambas variables, ya que ninguna representa el *timestamp* al completo.

Tras comprender la estructura interna de los mensajes y su composición, vamos a analizar la escritura y lectura de estos archivos.

3.2.1. Lectura del archivo bag

Para la lectura del archivo bag, se utilizó la librería *rosbag* [30]. Esta librería nos facilita la función *Bag()*, a la cual se le pasa por argumento el *path* del fichero a leer y nos devuelve un lector/editor del fichero bag.

Usando la función *read_messages()* sobre el lector/editor devuelto, se puede iterar sobre los mensajes del fichero rosbag. Además, para ver y elegir el tópico donde se encuentran los eventos, se utiliza la función *get_type_and_topic_info()* y se accede al campo *topics* de lo devuelto.

Una vez elegido el tópico, se leen los mensajes filtrados y se realiza la conversión al formato propio. Si los mensajes son de tipo *Event*, se guardan directamente en la lista; si por el contrario son de tipo *EventArray*, se itera por el array de eventos y se van añadiendo a la lista.

3.2.2. Escritura del archivo bag

Para la escritura del archivo, la librería *rosbag* [30] nos permite escribir en un bag con la función *write()*. Primero, se deben serializar los datos en un formato compatible con ROS . Para ello, se ha necesitado generar la clase *_Event*, la cual nos permite serializar un evento. Para generar esta clase, se ha utlizado *catkin*, y se ha seguido el siguiente tutorial [31].

Para asignarle a las instancias de esta clase *_Event* el *timestamp* del evento, se tuvo que usar la librería *rospy* [32], la cual nos permite acceder a la estructura time propia de ROS .

3.3. Archivos txt

El siguiente formato a analizar será el de los archivos de texto plano, los archivos txt. Como todo el mundo sabe, estos archivos acaban en *.txt* y contienen una secuencia de carácteres *American Standard Code for Information Interchange* (ASCII) .

En el caso de las cámaras de eventos, existe un formato presente en una gran cantidad de *datasets* para estos archivos. En esta configuración, se guarda un evento por línea, de tal manera que cada línea contiene la información necesaria para definir a un evento.

Cada línea está formada por una cadena de texto con los datos separados por un espacio (carácter 32 en ASCII). El orden de los parámetros que aparecen es el siguiente:

- *Timestamp* del evento en segundos. Si existe parte decimal, se separa de la parte entera con el carácter "."(carácter 46 en ASCII).
- Coordenada x del evento.
- Coordenada y del evento.
- Polaridad del evento, representada por el carácter 0 ó el carácter 1 (caracteres 48 y 49 en ASCII).

En la figura 3.1 se presenta un extracto del dataset [7] de la Universidad de Zúrich (UZH) en formato txt como ejemplo.

Cabe destacar que este dataset [7] contiene los datos también en formato bag, con lo que sirvió

0.001866000 118 31 1
0.001869001 167 158 0
0.001877000 139 172 0
0.001895001 35 40 1
0.001928000 187 74 1

Cuadro 3.1: Extracto del dataset de la Universidad de Zúrich en formato txt

para el testing. Más adelante se profundizará en esta parte en el capítulo 5.

En la figura 3.3 se presenta la tabla con las variables según su orden, correspondencia con variable común y tipo.

Posición en la cadena ASCII	Tipo de dato	Variable que representa
2	cadena de caracteres	coordenada x
3	cadena de caracteres	coordenada y
1	cadena de caracteres	*timestamp*
4	cadena de caracteres	polarización

Tabla 3.3: En esta tabla se especifica el formato de los eventos contenidos en los archivos txt.

Para finalizar la sección, se explicará como se ha implementado la lectura y escritura de este formato.

3.3.1. Lectura del archivo txt

Para la lectura del archivo txt, se abre el archivo con un objeto de edición de ficheros de Python mediante la función *open()* de Python. A continuación se procede a leer el archivo con la función *readlines()* y se analiza cada línea por separado para obtener y guardar la información de cada evento.

3.3.2. Escritura del archivo txt

Para escribirlo, se realiza también la apertura del archivo con la función *open()* y se va escribiendo con la función *write()* cada evento, uno por cada línea del archivo.

3.4. Archivos mat

Ahora se procederá a analizar el formato mat. Estos archivos son contenedores de información de datos que utiliza la aplicación MATLAB [26]. MATLAB es una de las aplicaciones para análisis de datos más grandes y usadas del mundo. En el programa desarrollado se soportan las versiones v4, v6 y v7 hasta v7.2.

Estos contenedores son capaces de almacenar diferentes tipos de datos como estructuras de datos o matrices. En el caso de las cámaras de eventos, todos los valores numéricos almacenados son de tipo **double**. Es decir, todas las variables informativas del evento (*coordendas x* e *y*, *timestamp* y *polarización*) son de tipo **double**.

3.4.1. Tres formatos de archivos .mat

A la hora de almacenar información sobre un conjunto de eventos captados por una cámara de eventos, existen diferentes formatos disponibles. En el programa se soportan tres, ya que son los comunes y más utilizados. A continuación se procederá a explicar sus formatos.

Una estructura con cuatro arrays

En esta configuración, se almacenan todos los datos en una estructura con cuatro variables. El nombre de esta estructura varía dependiendo del dataset escogido. Cada una de estas cuatro variables representa un array con la información de los eventos ordenados en el tiempo. Es decir, existen cuatro arrays diferentes, uno que almacena las *coordenadas x*, otro para las *coordenadas y*, otro para los *timestamp* y otro para las *polarizaciones*.

Por ejemplo, si queremos obtener el cuarto evento sucedido por orden cronológico, debemos acceder individualmente a la cuarta entrada de cada array de la estructura. Un ejemplo de dataset que utiliza este formato es el siguiente [13].

Además, en la tabla 3.4 se muestra el nombre, tipo y correspondencia de las variables en este formato.

Nombre de la variable	Tipo de dato	Variable que representa
No establecido, suele ser x	double	coordenada x
No establecido, suele ser y	double	coordenada y
No establecido, suele ser ts	double	*timestamp*
No establecido, suele ser pol	double	polarización

Tabla 3.4: En esta tabla se especifica el formato de los eventos contenidos en los archivos mat cuando su formato corresponde a una estructura con cuatro variables o a cuatro variables directamente.

Matriz N x 4

Con este formato, el archivo mat contiene una estructra con un solo campo, una matriz N x 4, siendo N el número de eventos guardados. Por cada fila de la tabla se guardan las cuatro variables que representan un evento. De tal manera que para acceder a un evento cualquiera, basta con acceder a la fila correspondiente de la matriz. El dataset [13] utiliza este formato.

En la tabla 3.5 se muestra el orden, tipo y correspondencia de las variables de este formato.

Orden en la matriz	Tipo de dato	Variable que representa
1	double	coordenada x
2	double	coordenada y
4	double	*timestamp*
3	double	polarización

Tabla 3.5: En esta tabla se especifica el formato de los eventos contenidos en los archivos mat cuando su formato corresponde a una matriz de tamaño N x 4.

Cuatro variables

En esta configuración de archivo, se almacenan cuatro variables, de manera similar al primer formato explicado en la sección 3.4.1. Cada variable contiene un array con los datos de los eventos, de tal manera que se accede a cada uno por separado para obtener la información total del evento. Como ejemplo, tenemos el dataset utilizado en esta aplicación [8].

Este formato sigue la misma tabla 3.4 de nombre, tipo y correspondencia que el formato de una estructura con cuatro variables.

3.4.2. Lectura del archivo mat

Para la lectura de los archivos *.mat* se ha utilizado las funciones *whosmat()* y *loadmat()* del paquete **scipy.io** [33]. La primera de estas funciones se utiliza para detectar el formato del archivo de entre los tres disponibles en la sección 3.4.1, ya que nos permite ver la estructura interna del archivo mat. La segunda nos permite cargar los datos del contedor a diccionarios y listas de python. De esta manera, es fácil realizar la lectura de los eventos almacenados.

3.4.3. Escritura del archivo mat

Al igual que en la lectura, se utiliza el paquete **scipy.io** [33], pero en este caso se emplea la función *savemat()*. El usuario decide la estructura que tendrá el archivo, eligiendo entre las tres disponibles en

la sección 3.4.1, a continuación el programa guarda los eventos con el formato escogido y mediante la función *savemat()* se guarda en el contenedor mat.

3.5. Archivos bin

En esta sección se mostrará el formato que contienen los archivos bin. Estos archivos contienen cualquier tipo de datos codificados en binario, por lo que tienen multitud de usos. Un archivo bin puede contener desde archivos multimedia hasta documentos de texto o programas [34].

En el campo de las cámaras de eventos, todos los archivos binarios siguen un formato común. En esta configuración, cada evento ocupa 40 bits, de tal manera que el archivo ocupa N x 40 bits, siendo N el número de eventos que contiene el archivo.

En la tabla 3.6 se expone la distribución de los bits según su correspondencia con el formato común.

Bits	Tipo de dato	Variable que representa
39-32	binario	coordenada x
31-24	binario	coordenada y
22-0	binario	*timestamp*
23	binario	polarización

Tabla 3.6: En esta tabla se especifica el formato de los eventos contenidos en los archivos bin.

Este formato sigue el modelo *big-endian*. Además, el *timestamp* viene definido en nanosegundos, con lo que habrá que realizar la conversión para pasarlo al formato común. También hay que tener en cuenta que las coordenadas pueden admitir valores entre 0 y 255, puesto que el campo asignado a estas variables tiene 8 bits de tamaño. En la figura 3.2 se muestra un ejemplo de como se expresaría un evento en estos 40 bits.

10000010 00100101 0 11001111000010000000000
Coordenada x $\rightarrow$ *10000010* $\rightarrow$ *130*
Coordenada y $\rightarrow$ *00100101* $\rightarrow$ *37*
Polarización $\rightarrow$ *0* $\rightarrow$ *False*
Timestamp $\rightarrow$ *11001111000010000000000* $\rightarrow$ *3392000 (nanosegundos)*

Cuadro 3.2: Ejemplo de un evento expresado en binario, tal y como se almacenaría en un archivo de tipo bin.

Este formato es común y estándar entre diversos *datasets* [14, 15].

En las siguientes dos subsecciones se profundizará en la lectura y ecritura de un archivo bin res-

pectivamente.

3.5.1. Lectura del archivo bin

Para la lectura del archivo bin, se ha utilizado las mismas funciones que en el formato texto 3.3.1. La principal diferencia es que en ese caso se lee la información codificada en ASCII , y en el formato bin, se ha de leer los datos byte a byte. Para ello, se abre el archivo en modo lectura binaria [35] y se procede a leer de cinco en cinco bytes. Una vez leídos y analizados, se obtienen los datos necesarios para instanciar al evento que representan y se guardan en la lista de eventos del formato común definido en 3.1.

3.5.2. Escritura del archivo bin

Es el mismo caso que en el de los archivos de texto 3.3.1. De manera similar al apartado de lectura de este formato 3.5.1, se debe abrir el archivo en escritura binaria [35], para poder escribir bytes en lugar de código ASCII . En este caso, se convierte el evento a almacenar según la codificación binaria ya explicada 3.6 y se procede a escribirlo en el archivo destino.

3.6. Archivos aedat

El último formato a exponer será el de los archivos *Address Event Data* (AEDAT) . Como su propio nombre indica, estos archivos se han desarrollado específicamente para almacenar los datos producidos por las cámaras de eventos.

Algunas versiones admiten información adiccional aparte de los propios eventos, como puede ser la información generada por el IMU por ejemplo. En este documento [36] se recogen todas las versiones con sus formatos correspondientes. En la aplicación se soportan las versiones *AEDAT 2.0*, *AEDAT 3.1* y *AEDAT 4.0*.

Las tres versiones admiten comentarios al principio del fichero. Estos comentarios están codificados en ASCII y cada comentario debe ocupar una línea y empezar por el **símbolo #** (número ASCII 35), y acabar en **salto de línea** (número ASCII 10).

Puede haber tantos comentarios como se desee, pero siempre al principio del fichero debe aparecer un primer comentario indicando la versión del archivo. En la tabla 3.7 se expone el valor de este comentario para cada versión.

En las siguientes subsecciones, se explicará el formato de cada una de estas versiones. Finalmente, se profundizará en la implementación de la lectura y escritura de los archivos AEDAT según su

Versión	Contenido del primer comentario
AEDAT 2.0	#!AER-DAT2.0
AEDAT 3.1	#!AER-DAT3.1
AEDAT 4.0	#!AER-DAT4.0

Tabla 3.7: En esta tabla se especifica el valor del primer comentario para cada versión de AEDAT

versión.

3.6.1. AEDAT 2.0

En este formato, después de la sección de comentarios, cada evento va codificado en 8 bytes propios. Estos 8 bytes se dividen en 2 partes diferentes y están codificados en *big-endian*:

Address: Los primeros 4 bytes contienen codificado en su interior la *polarización* y las coordenadas *x* e *y*. A esta sección se la conoce como *address*.

Coordenada X: Desde el bit 8 hasta el 14, ambos incluidos.

Coordenada Y: Desde el bit 1 hasta el 7, ambos incluidos.

Polarización: Solo el bit 15. Si es 1, la polarización es True; si es 0, False.

Timestamp: Los restantes 4 bytes describen el *timestamp* del evento. Esta variable se almacena en nanosegundos.

Como se puede observar en el *address*, tanto el bit 0 como los bits del 16 al 31 no tienen valor asignado. Esto se debe a que en estos bits se almacenan datos adiccionales al evento como podría ser los datos generados por el IMU o la calibración de la cámara por ejemplo.

En el conversor desarrollado, estos datos tienen valor 0 para conservar igualdad entre formatos, ya que el resto de formatos no admiten está característica. Además, las coordenadas deben tener un valor entre 0 y 127, puesto que su campo solo tiene 7 bits de tamaño.

En la figura 3.3 se muestra un ejemplo del address contenido en estos archivos.

0000000000000000 1 0111001 1110010 0
Coordenada x $\rightarrow$ *0111001* $\rightarrow$ *57*
Coordenada y $\rightarrow$ *1110010* $\rightarrow$ *114*
Polarización $\rightarrow$ *1* $\rightarrow$ *True*

Cuadro 3.3: Ejemplo de un evento expresado en binario, tal y como se almacenaría en un archivo de tipo AEDAT versión 2.0.

En la tabla 3.8 se hace hincapié en el formato de la versión 2.0 de AEDAT , incluyendo los valores correspondientes al formato común 3.1.

Bits	Tipo de dato	Variable que representa
8-14	binario	coordenada x
1-7	binario	coordenada y
32-63	binario	*timestamp*
15	binario	polarización

Tabla 3.8: En esta tabla se especifica el formato de los eventos contenidos en los archivos AEDAT 2.0.

3.6.2. AEDAT 3.1

En esta versión, los eventos se almacenan en contenedores separados, cada uno con sus propias cabeceras. Estas cabeceras ocupan 28 bytes y la principal cabecera que nos atañe es la que se sitúe entre el byte 20 y el 23. Esta cabecera especifica el número de eventos que almacena el contenedor al que acompaña.

El resto de cabeceras almacenan datos como la calibración de la cámara, por lo que no se tienen en cuenta para la conversión. A la hora de escribirlos se almacenan como cabeceras vacías (todo valor 0).

Una vez leído el número de eventos que guarda el contenedor, se procede a leer cada evento por separado. Cada evento ocupa un total de 8 bytes almacenados en formato *little-endian*. Su configuración es la siguiente:

Data: Los primeros 4 bytes contienen codificado en su interior la *polarización* y las coordenadas *x* e *y*. A este tramo se le conoce como *data*.

- **Coordenada X:** Desde el bit 17 hasta el 31, ambos incluídos.
- **Coordenada Y:** Desde el bit 2 hasta el 16, ambos incluídos.
- **Polarización:** Solo el bit 1. Si es 1, la polarización es True; si es 0, False.

Timestamp: Los restantes 4 bytes describen el *timestamp* del evento. Esta variable es almacenada en nanosegundos.

El bit en la posición 0 es el bit de validación y su valor siempre debe ser 0. Se debe tener en cuenta que las coordenadas x e y deben estar comprendidas entre 0 y 32767, puesto que cada campo ocupa 15 bits.

En el recuadro 3.4 se mostrará un ejemplo de escritura del evento en formato binario para esta versión, omitiendo el timestamp. Para facilitar la comprensión, se muestra el ejemplo en formato *big-*

endian.

000000101100101 000001000110010 1 0
Coordenada x $\rightarrow$ *000000101100101* $\rightarrow$ *357*
Coordenada y $\rightarrow$ *000001000110010* $\rightarrow$ *562*
Polarización $\rightarrow$ *1* $\rightarrow$ *True*

Cuadro 3.4: Ejemplo de un evento expresado en binario, tal y como se almacenaría en un archivo de tipo AEDAT versión 3.1.

Para finalizar la subsección, se muestra en la tabla 3.9 el formato que siguen los 64 bits para almacenar un evento.

Bits	Tipo de dato	Variable que representa
17-31	binario	coordenada x
2-16	binario	coordenada y
32-63	binario	*timestamp*
1	binario	polarización

Tabla 3.9: En esta tabla se especifica el formato de los eventos contenidos en los archivos AEDAT 3.1.

3.6.3. AEDAT 4.0

La versión 4.0 de AEDAT utiliza los Flatbuffers de Google [37] para serializar los datos. En este formato, el contenido del fichero se divide en diversos paquetes. Los eventos se guardan en el paquete de nombre *events*.

El archivo contiene dos tipos de *headers*. El primero se encuentra al comienzo del archivo y añade información como la compresión utilizada o el esquema que siguen los paquetes. El segundo se encuentra al inicio de cada paquete indicando su tamaño y su offset. Estos *headers* se encuentran definidos en el siguiente repositorio [38].

El paquete *events* contiene todos los eventos serializados. Lamentablemente, todavía no existe un esquema para serializar los datos de los eventos. Por este motivo, no se puede especificar el formato para este tipo de archivos.

3.6.4. Lectura del archivo AEDAT

Tanto en la versión 2.0 como en la versión 3.1, se leen los datos con las funciones *open()* y *read()* de Python al igual que en el formato txt 3.3.1 y en el formato bin 3.5.1 , más parecido a este último pues se lee el fichero en modo binario.

En la **versión 2.0**, se leen de 8 bytes en 8 bytes los datos, analizando cada sección para obtener los datos del evento que representan. Una vez analizados los datos, se procede a instanciar y almacenar el evento correspondiente en la lista de eventos.

En cuanto a la **versión 3.1**, es un poco más complejo de analizar. Se leen las cabeceras por contenedor para analizar cuantos eventos guarda ese contenedor. Una vez leído el número de eventos almacenados, se van leyendo en secciones de 8 bytes para obtener los eventos guardados en el contenedor. Una vez se ha leído todo el paquete de eventos, se procede a leer de nuevo las cabeceras del siguiente contenedor para analizarlo y guardarlo en la lista de eventos. Si no existen cabeceras, el algoritmo finaliza y devuelve la lista con los eventos almacenados. En la figura 3.1 se muestra el algoritmo seguido para la lectura de esta versión.

```
input : Archivo AEDAT 3.1 a leer
output: Lista de eventos que contiene el archivo AEDAT 3.1
Lectura de los comentarios;
while True do
    header ← read(28);
    if len(header) == 28 then
        num_events ← header[20:23];
        for i ← 0 to num_events do
            event ← read(8);
            analized_event ← analizeEvent(event);
            event_list += analized_event;
        end
    end
    else
        break;
    end
end
return event_list;
```

Algoritmo 3.1: Pseudocódigo que hace la función de lectura de los eventos en un archivo AEDAT 3.1

Para la **versión 4.0**, se ha utilizado la librería *aedat* de Python [39]. Este paquete nos ofrece la función *Decoder()*, la cual nos permite acceder fácilmente a los paquetes contenidos dentro del fichero. Para obtener los eventos, se debe iterar sobre el paquete de nombre *events*, el cual contiene los eventos deserializados.

3.6.5. Escritura del archivo AEDAT

Para la escritura en las versiones 2.0 y 3.1, se sigue el mismo procedimiento que en los formatos txt 3.3.2 y bin 3.5.2, siendo más cercano a este último.

En la **versión 2.0** se itera sobre los eventos a guardar y se van escribiendo según su formato de 8 bytes. Para ello se utiliza la función *write()* en modo binario [35], lo que nos permite escribir estos 8 bytes de golpe.

En la **versión 3.1** se almacenan todos los eventos en un mismo contenedor. Primero se escriben los *headers* del contenedor, teniendo todas estas el valor 0; excepto la cabecera que indica el número de eventos cuyo valor lo ajustamos en función del tamaño de la lista de eventos. También se utiliza la función *write()* al igual que en la versión 2.0.

Para la **versión 4.0** no se ha podido implementar la escritura, puesto que todavía no se han definido los esquemas para la serialización de los datos. Sin estos esquemas, no se pueden serializar los datos con los Flatbuffers de Google.

4

Desarrollo

En este capítulo se profundizará en el propio desarrollo del programa, explicando su implementación y sus principales características.

4.1. Módulos

El programa desarrollado se divide en diversos módulos para facilitar la implementación y la depuración del sistema. Existen cinco módulos en total más el programa principal:

Módulo de Conversores: Contiene la funcionalidad principal para convertir los archivos. Se implementa la lectura y la escritura de todos los formatos.

Módulo de Interfaz de Usuario: En este módulo se han desarrollado tres IU para el uso del programa.

Módulo de Configuración: Gracias a este módulo se pueden configurar los datos usados en las conversiones para facilitar su automatización.

Módulo de Parseo de Argumentos: Este módulo implementa la funcionalidad vinculada al parseo de los argumentos de entrada del programa.

Módulo de Utilidades: En este módulo encontraremos funcionalidades que nos permiten simplificar el código.

Main del programa: Este archivo es el encargado de correr el hilo principal de la aplicación.

Tanto el módulo de conversores, como el de parseo de argumentos, como el de utilidades proporcionan funciones que el main, la IU y el módulo de configuración utilizan.

En contraparte a estos módulos, se hallan tanto el módulo de configuraciones como el módulo de la interfaz de usuario, los cuales implementan el patrón Singleton 4.1.1, permitiéndonos acceder a una clase común y única desde cualquier punto del programa. De esta manera se pueden utilizar las funcionalidades de ambas clases sin tener que administrarlas como argumento al resto de funciones.

4.1.1. Patrón Singleton

Este patrón de diseño nos permite definir una clase la cual solo se va a instanciar una vez. De esta manera se garantiza la existencia de un solo objeto de su tipo. Además debe proporcionar un punto de acceso desde cualquier parte del código hacia esa instancia, de tal manera que cada vez que se llame al constructor de la clase, esta nos devuelve la instancia ya existente. En caso de no existir, se instanciaría el objeto y se asignaría este como la instancia por defecto [40].

Para desarrollar este patrón de creación en Python se ha seguido la guía de refactoring [41]. Según este método, se define una clase que sobreescribe el método __call__ para que nos devuelva siempre la instancia ya existente. Esta clase creada (de nombre *SingletonMeta* en nuestro caso) se deberá asignar como metaclase [42] de cualquier clase que queramos que siga el patrón Singleton.

4.1.2. Clases adiccionales

Fuera de los módulos principales del programa, se han desarrollado dos clases adiccionales para el correcto uso del módulo de conversores. Estas dos clases son:

EventClass: Clase usada como formato propio. Esta clase se utiliza para instanciar eventos compatibles con el resto de formatos. Es el nexo de unión entre las diferentes configuraciones de archivo. Contienen además comprobación de tipos para facilitar el control de errores. Esta clase viene más explicada con detalle en la sección 3.1.

EventRosbag: Gracias a esta clase se puede realizar la escritura del archivo bag. Esta clase está explicada en la subsección 3.2.2.

4.2. Conversores

En esta sección se explicará el funcionamiento del módulo de conversores así como su implementación y desarrollo.

Para comenzar, es importante destacar el archivo que contiene la funcionalidad principal, el fichero **mainConverter.py**. Este fichero contiene la función *convert()*, la cual se encarga de recibir el fichero de entrada junto a su tipo, y devolverlo escrito en el formato de salida requerido y en el fichero especificado. Para ello, primero lee el archivo con el conversor necesario, obteniendo la lista de eventos que estaban almacenados en ese fichero. A continuación, escribe esta lista de eventos en el archivo de salida con el formato requerido.

La función *convert()* recibe como parámetros de entrada el tipo de entrada y de salida, y el *path* del fichero de entrada y del fichero de salida.

Para realizar la escritura y lectura de los ficheros, la función *convert()* llama a las funciones correspondientes del módulo según el formato necesario. En la tabla 4.1 se pueden observar las funciones de escritura y lectura vinculadas con cada tipo de configuración de archivo.

Formato	Función de lectura	Función de lectura
bag	rosbagToAbstract	abstractToRosbag
txt	textToAbstract	abstractToText
mat	matlabToAbstract	abstractToMatlab
bin	binToAbstract	abstractToBin
AEDAT	aedatToAbstract	abstractToAedat

Tabla 4.1: En esta tabla se especifican las funciones de lectura y escritura del módulo de conversores según su formato.

Las funciones de lectura reciben como parámetro el *path* del fichero de entrada; y las funciones de escritura reciben dos parámetros, la lista de eventos a escribir y el *path* del fichero de salida.

En cuanto a los archivos AEDAT , las funciones de escritura y lectura simplemente llaman a los sub-conversores desarrollados para cada versión. En la tabla 4.2 se puede observar las funciones de lectura y escrituras implementadas para cada versión de AEDAT . Hay que tener en cuenta que la función *abstractToAedat4()* no está implementada, para más información ver el apartado 3.6.5.

Versión de AEDAT	Función de lectura	Función de lectura
2.0	aedat2ToAbstract	abstractToAedat2
3.1	aedat3ToAbstract	abstractToAedat3
4.0	aedat4ToAbstract	abstractToAedat4

Tabla 4.2: En esta tabla se especifican las funciones de lectura y escritura de la funcionalidad desarrollada para AEDAT.

Cualquiera de estas funciones puede ser utilizada externamente por otro desarrollador de manera simple. Además de la función *convert()*, el fichero **mainConverter.py** contienen dos funciones auxiliares para que cualquier desarrollador pueda incorporarlas fácilmente a su código.

getEventsFromFile: Esta función obtiene los eventos del fichero pasado por parámetro. Esta función recibe como parámetros el tipo de fichero de entrada y el *path* del fichero de entrada.

putEventsInFile: Esta función escribe los eventos pasados por parámetro en el archivo destino con el tipo especificado. Recibe por parámetro la lista de eventos, el tipo de salida y el *path* del fichero de salida.

4.3. Interfaz de usuario

Para este programa, se han desarrollado tres IU diferentes. Para seleccionar que interfaz se desea utilizar, se debe indicar en los argumentos de entrada del programa 4.5. Por defecto está seleccionada la IU transparente.

Las tres interfaces desarrolladas heredan de una clase abstracta llamada *InterfaceUI*. Para ello, utilizamos la librería **abc** de Python. Esta librería nos permite implementar una clase que funcione como plano para las clases que hereden de ella. La clase abstracta desarrollada contiene siete métodos que deben ser sobreescritos por toda clase que herede de esta. Estos métodos son los siguientes:

initialWindow(): Llama a la funcionalidad principal del programa (la función *convert()* del módulo de conversores 4.2). Es la función que se ejecuta al iniciar la aplicación.

showMessage(): Este método genera un mesaje informativo para notificar al usuario. El mensaje se le pasa por parámetro.

sumProgress(): Esta función se encarga de mostrar la barra de progreso. Cada vez que se llama a este método, la barra de progreso se actualiza.

simpleInput(): Este método se encarga de pedir y recoger un dato del usuario.

chooseWindow(): Se ofrece al usuario multiples opciones, para que este escoja la opción que desee.

multiInputsWindow(): Se le ofrece al usuario multiples opciones, teniendo que darles a todas un dato de entrada.

errorWindow(): Con esta función se imprime un mensaje de error en caso de producirse. Además, se muestra la *traceback* que lleva la excepción por detrás.

Para alojar estas tres clases, se ha desarrollado una clase adiccional, de nombre *UI*. Está sigue el patrón Singleton 4.1.1, de tal manera que en su primera invocación, genera la interfaz requerida y la guarda en una variable de nombre *objectUI*. Con esta técnica, se puede acceder desde cualquier punto del código a la interfaz en uso.

4.3.1. Interfaz de usuario transparente

Esta interfaz está diseñada para usuarios que no desean ver información sobre la ejecución del programa. Con esta opción habilitada, todos los datos necesarios deben ser pasados por parámetros del programa. Una vez se empiece a ejecutar el programa, no se informará de nada al usuario. Solo se imprimirá por la terminal información en caso de que hubiese ocurrido un error. En este caso, se imprimirá por pantalla la excepción ocurrida junto al *traceback* de esta misma.

Para esta interfaz, se ha desarrollado la clase *TransparentUI*. Esta clase solo implementa el método

errorWindow(), el resto de métodos se implementan con funcionalidad vacía. Esto se debe a que estos métodos informan al usuario del estado del programa en ejecución, y en esta interfaz esta información solo se muestra en caso de error.

A la hora de ejecutar este programa, se debe iniciar la aplicación introduciendo como parámetros todos los necesarios para que el conversor funcione. Para más detalles ver la matriz de argumentos por interfaz, cuadro 4.3.

4.3.2. Interfaz de usuario basada en terminal

Con esta interfaz, el usuario podrá ejecutar el programa y recibir información sobre este a través de la terminal donde se esté ejecutando. Este programa preguntará al usuario por los datos necesarios para realizar la conversión. Sin embargo, necesita tener como entrada de programa tanto el *path* del fichero de entrada como el *path* del fichero de salida (ver tabla 4.3). Si recibe algún parámetro adiccional, será usado en la conversión.

Para esta interfaz, se ha implementado la clase *TerminalUI*. Esta clase implementa todos los métodos que hereda. Para darle un tono más amable, se ha utilizado la funcionalidad *colors* del módulo de utilidades 4.6.1. De esta manera, se pueden imprimir mensajes con colores por la terminal.

Para poder usar esta interfaz, se debe introducir el parámetro - *-use_terminal_UI*.

En la figura 4.1 se puede observar un ejemplo de conversión utilizando esta interfaz.

4.3.3. Interfaz de usuario gráfica

Esta interfaz es la más dirigida hacia el usuario inexperto, pues permite introducir todos los datos necesarios para la conversión de manera simple y guiada. Esta IU no requiere que la aplicación reciba ningún parámetro de entrada. En caso de recibir algún parámetro, se usará por defecto en la conversión.

Al igual que en la IU basada en terminal, la clase desarrollada implementa todos los métodos heredados. Esta clase se llama *GraphicUI* y para su desarrollo se ha utilizado la librería PySimpleGUI [43]. Este paquete permite el uso de interfaces gráficas muy vistosas y fáciles de implemetar.

Para utilizar esta interfaz, se debe usar el parámetro - *-use_graphic_UI*.

Nada más iniciar la aplicación, aparecerá una pantalla similar a la mostrada en la figura 4.2. Como se puede observar, esta pantalla nos permite añadir todos los parámetros necesarios para la conversión de un archivo. También dispone de un botón para resetear todos los valores que aparecen en la ventana. Una vez se haga click en el botón de *CONVERT*, se empezará la conversión guiada.

```
---STARTING CONVERSION---
Starting to read bin file
-0%-
-10%-
-20%-
-30%-
-40%-
-50%-
-60%-
-70%-
-80%-
-90%-
-100%-
Finishing reading the bin file
Starting to write bag file
Introduce the name of the topic where the events are going to be write: /dvs/event
-0%-
-10%-
-20%-
-30%-
-40%-
-50%-
-60%-
-70%-
-80%-
-90%-
-100%-
Finishing writing the bag file
---CONVERSION FINISHED---
```

Figura 4.1: Figura de ejemplo de uso de la interfaz de usuario basada en la terminal.

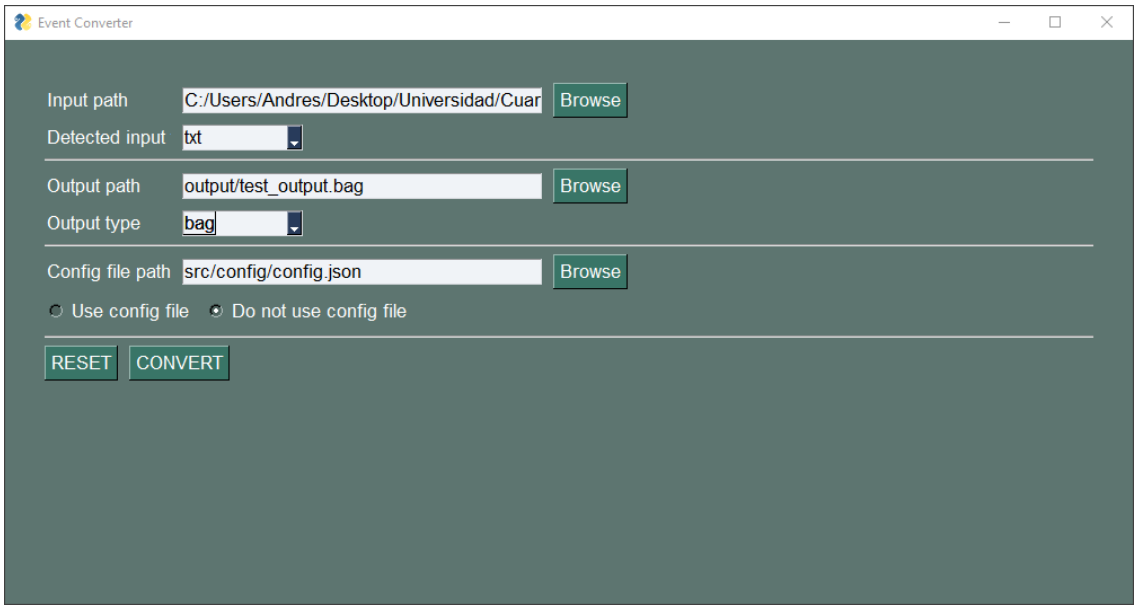

Figura 4.2: Ventana principal de la interfaz gráfica de la aplicación. Se observa que todos los parámetros pueden ser introducidos desde aquí.

Después de iniciar la conversión, el programa irá mostrando por pantalla mensajes para visualizar el progreso de la conversión. En caso de que la lectura o escritura necesite algún dato extra para la conversión, le aparecerá una ventana emergente pidiendo al usuario la información requerida. Se puede observar un ejemplo de esto en la figura 4.3.

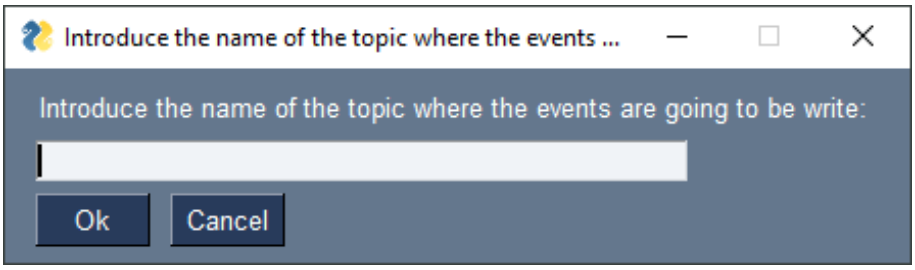

Figura 4.3: Ventana emergente de la interfaz gráfica de la aplicación. En este caso es de tipo entrada simple, ya que solo le pide un dato al usuario.

Una vez empezada la conversión con todos los parámetros ya introducidos, se procederá a la lectura y escritura de los ficheros. Para visualizar estos procesos, se ha implementado una barra de progreso que nos indica el tiempo restante estimado. Además, nos permite cancelar la operación en cualquier instante de la conversión. En la figura 4.4 se puede observar la ventana que contiene esta barra de progreso.

Una vez finalizada la conversión, se informará al usuario con una ventana emergente y se volverá a la pantalla inicial 4.2. En caso de error, se informará al usuario con una ventana de error y se cerrará el programa. Esta ventana nos permite ver el *traceback* que ha causado el error. Por ejemplo, en la

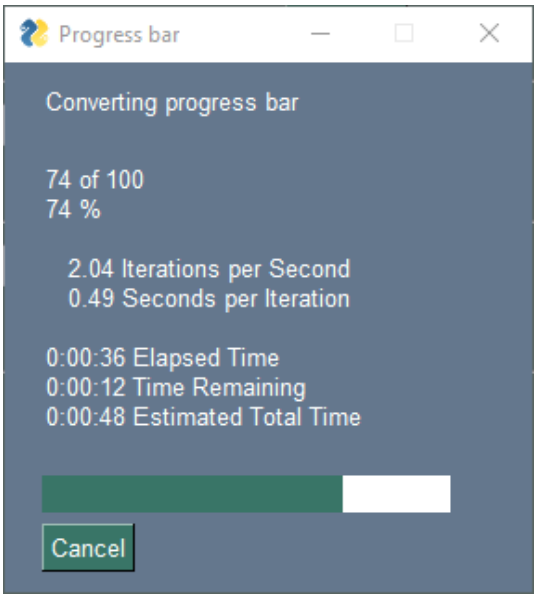

Figura 4.4: Barra de progreso de la interfaz gráfica de la aplicación. A parte de la representación gráfica del progreso, también muestra el tiempo restante estimado de la conversión.

figura 4.5 se muestra la ventana de error tras introducir coordenadas más grandes de las que admite el formato bin.

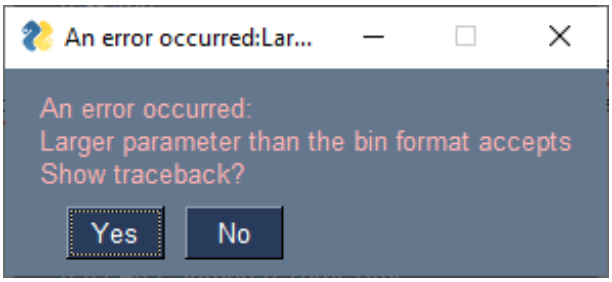

Figura 4.5: Ventana de error de la interfaz gráfica de la aplicación. Esta ventana permite visualizar el traceback de la excepción recogida.

4.4. Archivo de configuración

En esta sección se profundizará en el módulo de configuración. Este paquete nos permite guardar datos que el conversor utilizará por defecto. De esta manera se puede automatizar el proceso de conversión, facilitando la conversión de múltiples archivos.

Este módulo contiene dos funcionalidades principales. En las siguientes subsecciones se explicará cada una de estas.

4.4.1. Archivo de constantes

Toda esta funcionalidad está implementada en el archivo **constants.py**. Este fichero contiene once variables que el sistema utiliza por defecto. Se recomienda no modificar este fichero, pero en caso de necesitarlo, se muestra a continuación una lista con las descripciones de cada variable:

ADMITTED_TYPES: Lista donde se almacenan los tipos de archivos admitidos. Esta lista almacena solo las extensiones de los archivos admitidos.

CONFIG_PATH: *Path* del fichero de configuración para conversiones, ver apartado 4.4.2.

NUM_PROGRESS_BAR: Indica el número de pasos que la barra de progreso de la IU gráfica ejecuta. Se recomienda 100 para conversiones pequeñas y 1000 para conversiones más grandes.

DATA_PATH: En esta variable se guarda el *path* de la carpeta de datasets. Este path es usado en el apartado de testing 5 para facilitar el uso de diferentes datasets.

OUTPUT_DEFAULT: Nombre del archivo de salida por defecto. En caso de no especificar el *path* del fichero de salida, se utiliza este valor. No debe incluir la extensión del archivo pero si el punto final.

MATLAB_STRUCT_NAMES: Lista que guarda los cuatro nombres por defecto que tienen las variables de MATLAB de los formatos *1 estructura* 3.4.1 y *4 estructuras* 3.4.1. No se utilizan en la conversión, solo para informar al usuario de que variables del formato común 3.1 corresponden a cada estructura, es decir, solo tienen efecto para la IU .

MATLAB_TYPES_ADMITTED: Lista con los tipos que se aceptan del formato MATLAB , explicados en la sección 3.4.

AEDAT_ACCEPTED_VERSIONS: Versiones que se aceptan del formato AEDAT , mostrados en la sección 3.6.

INITIAL_COMMENTS_AEDAT2: Lista con los comentarios que se escribirán por defecto en AEDAT 2.0. El primer comentario debe ser "#!AER-DAT2.0\n".

INITIAL_COMMENTS_AEDAT3: Lista con los comentarios que se escribirán por defecto en AEDAT 3.1. El primer comentario debe ser "#!AER-DAT3.1\n".

INITIAL_COMMENTS_AEDAT4: Lista con los comentarios que se escribirán por defecto en AEDAT 4.0. El primer comentario debe ser "#!AER-DAT4.0\n".

Como se puede intuir para los comentarios de AEDAT , todo comentario añadido debe comenzar por por el símbolo # y acabar en \n (salto de línea).

4.4.2. Archivo de configuración

Esta funcionalidad da la posibilidad de configurar las conversiones para automatizarlas. Se divide en dos archivos diferentes, el lector del archivo de configuración (**config.py**) y el propio archivo de configuración (**config.json**).

El primero de ellos contiene una clase que sigue el patrón Singleton 4.1.1. De esta manera, al instanciar el objeto por primera vez, se lee el archivo de configuración con la librería para la lectura de archivos JSON de Python [44]. Una vez leído los datos, se guardan en la instancia y se usan en el resto de la aplicación. En caso de no querer utilizar esta característica, se instancia una clase sin datos.

El archivo de configuración se ha codificado en JSON para facilitar su modificación. Además, la aplicación permite el uso de diversos archivos de configuración, en la sección de argumentos de entrada se explica más al respecto 4.5. En la figura 4.6 se puede observar un ejemplo del archivo de configuración.

Figura 4.6: Figura que muestra el archivo de configuración de conversión por defecto.

Los formatos que necesitan configuración extra para su conversión son los archivos bag, mat y aedat. A continuación, se procederá a explicar cada variable del archivo de configuración por separado.

rosbag: Variables para la conversión de archivos bag.

topic: Esta variable indica el tópico que se usará para la lectura y escritura de archivos bag.

matlab: Variables para la conversión de archivos mat.

default: Indica que formato de mat se escribirá por defecto. Admite los siguientes valores: '1 struct', 'Matrix nx4' y '4 structs'.

1 struct: Almacena variables para el formato con una estructura.

struct_name: Nombre de la estructura principal. Se usa tanto en lectura como escritura.

names: Nombres de las cuatro variables que contiene la estructura. El orden que sigue es: coordenada x, coordenada y, polarización, *timestamp*.

Matrix nx4: Almacena variables para el formato con una matriz N x 4.

struct_name: Nombre de la estructura principal. Se usa tanto en lectura como escritura.

indexes: Índices de cada variable de la matriz. La primera posición representa la columna de la coordenada x, la segunda la de la coordenada y, la tercera de la polarización y la cuarta del *timestamp*.

4 structs: Almacena variables para el formato con cuatro estructuras.

names: Nombres de las cuatro variables. El orden que sigue es: coordenada x, coordenada y, polarización, *timestamp*.

aedat: Variables para la conversión de archivos aedat.

version: Indica la versión de aedat que se usará por defecto en la escritura. Admite los siguientes valores: 'aedat 2.0', 'aedat 3.1' y 'aedat 4.0'.

4.5. Argumentos de entrada

Este módulo contiene la funcionalidad necesaria para el parseo de argumentos de programa.

Para ello, se ha utilizado la librería **argparse** de Python [45]. Gracias a esta librería se ha podido implementar un sencillo parseo de los argumentos del programa. Existen dos tipos de argumentos diferentes, los posicionales y los no posicionales.

Los argumentos **posicionales** deben colocarse según el orden establecido. En el programa existen dos de este tipo:

input_file: Este argumento guarda el *path* del fichero de entrada, debe ir en primera posición.

output_file: Guarda el *path* del fichero de salida, debe ir en segunda posición. En caso de no utilizar este parámetro, se utilizará la salida por defecto, ver apartado 4.4.1.

En cuanto a los argumentos no posicionales, no necesitan seguir ningún orden en específico. Los argumentos no posicionales admitidos son:

- -input_type: Este argumento debe ir acompañado de una cadena de texto que indica el tipo de formato con el que se va a leer el fichero de entrada, independientemente de su extensión.

- -output_type: Igual que el *- -input_type* pero con el archivo de salida.

- -config_path: Indica el *path* del archivo de configuración. La cadena de texto que lo acompaña hace referencia al fichero que se cargará como archivo de configuración. Gracias a este parámetro se pueden usar diversos archivos de configuración.

- -use_config: Si este parámetro se utiliza, se usará el fichero de configuración. Sino, la IU correspondiente le preguntará al usuario por los datos necesarios para la conversión.

- -use_graphic_UI: Utilizar este argumento si se desea utilizar la IU gráfica.

- -use_terminal_UI: Utilizar este argumento si se desea utilizar la IU basada en terminal.

Según la interfaz de usuario escogida, algunos parámetros serán obligatorios y otros opcionales. En la figura 4.3 se muestra la matriz de correspondencia entre argumentos e IU , indicando los parámetros necesarios para cada interfaz.

Argumento	UI transparente	UI basada en terminal	UI gráfica
input_file	X	X	O
output_file	O	O	O
- -input_type	O	O	O
- -output_type	X	X	O
- -config_path	O	O	O
- -use_config	X	O	O
- -use_graphic	I	I	X
- -use_terminal	I	X	I

Tabla 4.3: En esta tabla se muestra los argumentos que son obligatorios según la interfaz utilizada. Una X indica que ese parámetro es obligatorio para esa interfaz, una O indica que es opcional, y una I indica que es incompatible.

Para acabar esta sección, se muestra un ejemplo de ejecución del programa en el cuadro 4.1. En esta figura, se ha ejecutado con interfaz gráfica, usando el archivo de configuración, con archivo de entrada *input_file.bag*, tipo de salida *mat* y utilizando el fichero de configuración *config/config.json*.

python3 main.py *input_file.bag - -use_terminal_UI - -use_config - -output_type mat - -config_path config/config.json*

Cuadro 4.1: En esta figura se muestra un ejemplo de ejecución del programa con diversos argumentos.

4.6. Módulo de utilidades

El módulo de utilidades contiene diversas funciones que han ayudado a simplificar el código del programa. Estas funciones se han desarrollado en dos ficheros independientes.

4.6.1. Fichero colors.py

Este módulo ofrece cinco funciones para imprimer texto de colores por la terminal. Para ello, se imprime una secuencia de caracteres (conocidos como código escape *American National Standards Institute* (ANSI)) que indican a la terminal el cambio de código de color [46]. Los colores disponibles son: azul, verde, amarillo, rojo y blanco.

4.6.2. Fichero utils.py

Este fichero aporta siete funciones diferentes totalmente independientes unas de otras. Además, este archivo contiene la metaclase SingletonMeta usada para el patrón Singleton 4.1.1. Las funciones desarrolladas en este archivo son:

checkPathAndType: Checkea si el *path* y el tipo de formato pasado por parámetro son aceptados por el programa. En caso de error en uno de ellos, se lanza una excepción.

combine: Se pasa por parámetro la parte entera y la parte decimal de un número, y se combinan para generar un número con parte decimal.

nsecsToSecs: Convierte el dato pasado por parámetro de nanosegundos a segundos.

secsToNsecs: Convierte el dato pasado por parámetro de segundos a nanosegundos.

getExtension: Obtiene la extensión del archivo pasado por parámetro.

addExtension: Devuele el *path* del fichero pasado por parámetro con la nueva extensión también pasada por parámetro.

getNumProgress: Obtiene el número de pasos que la barra de progreso debe realizar según el número de eventos pasado por parámetro.

4.7. Main

En esta sección se profundizará en el funcionamiento del archivo principal del programa. Toda la funcionalidad principal está programada en el fichero **main.py**. Este archivo contiene la función *main()* y la llamada a esta. Esta función realiza el arranque del programa, su código aparece expuesto en la página 36. Como se puede observar, el hilo de ejecución del programa es el siguiente:

1.– Se realiza el parseo de los argumentos de entrada el programa. En caso de error, se termina la ejecución.

2.– Si el archivo de configuraciones se utiliza, llamar al constructor de la clase *Config* para cargar los datos necesarios.

3.– A continuación, se inicia la interfaz de usuario correspondiente. Si no es ni gráfica ni basada en terminal, no hace falta inicializar la interfaz.

4.– Por último, se lanza la aplicación llamando a la función *initialWindow()* de la interfaz correspondiente. Esta llamada se realiza dentro de try-catch, de tal manera que se pueda recoger cualquier excepción ocurrida en el código desde el programa principal. La excepción recogida se muestra al usuario con la interfaz en uso.

Código 4.1: En esta figura se muestra el contenido de la función main del programa.

```
1  def main():
2      # Args parse.
3      input_file, output_file, input_type, output_type, use_config, config_path, ui_type =
           parseArguments()
4
5      # Init config features.
6      if use_config:
7          Config(config_path)
8
9      # Create UI.
10     if ui_type == "graphic":
11         UI("graphic")
12     elif ui_type == "terminal":
13         UI("terminal")
14
15     # Init UI.
16     try:
17         UI().objectUI.initialWindow(convert, input_file, output_file, input_type, output_type,
               use_config, config_path)
18     except Exception as e:
19         UI().objectUI.errorWindow(e)
```

5

Pruebas

En este capítulo se profundizará en las pruebas realizadas para garantizar el correcto funcionamiento del sistema desarrollado.

Para la realización de estas pruebas, se han implementado diversas funciones en el archivo *testUtils.py*. Estos métodos nos permiten comprobar el correcto funcionamiento de los conversores utilizando datasets externos para ello. A continuación, se explicará cada una de las ocho funciones implementadas.

equalEvent: Recibe dos eventos por parámetro. Devuelve verdadero si son iguales y falso en caso contrario.

equalEventList: Comprueba si las dos listas de eventos pasadas por parámetros son iguales. En caso de desigualdad, imprime el primer evento que ha detectado diferente.

getListOfFiles: Obtiene todos los ficheros que se encuentren dentro de la carpeta pasada por parámetro. Inlcuye subdirectorios.

generateRandomEvents: Genera una lista de eventos aleatorios. El tamaño de la lista se le pasa como parámetro.

generateRandomFiles: Genera una lista de eventos aleatorio y lo guarda en todos los formatos disponibles.

testFileAndType: Función que comprueba la conversión a un tipo pasado por parámetro. Se comprueba usando un archivo, cuyo *path* es pasado también como parámetro.

testTypes: Se comprueba la conversión de dos tipos pasados por parámetros. Se genera una lista de eventos aleatoria y se escriben en los dos formatos. Acto seguido, se leen ambos archivos y se chequea que las listas de eventos leídas son iguales.

testShuffle: Se realiza un test por cada fichero existente en la carpeta de datasets 4.4.1. Este test se realiza con la función *testFileAndType()* y lo comprueba con un tipo aleatorio de entre los admitidos.

Tanto la función *testFileAndType()* y como la función *testTypes()* se basan en la misma idea, leer los eventos almacenados en el archivo y escribir estos eventos de nuevo en otro formato o en el mismo

formato. Si el sistema funciona correctamente, al leer los archivos se deberá obtener la misma lista de eventos en ambos ficheros. De esta manera se puede verificar que el conversor es capaz tanto de leer como de escribir sin perder datos o precisión en el proceso.

Para las pruebas y el desarrollo, se han utilizado diversos datasets que diferentes grupos de investigación han colgado en la red, por ejemplo: [8, 13–18]. Muchos de estos datasets aportan los datos en diferentes formatos [7, 19]. Gracias a esto se ha podido chequear más fácilemente el correcto funcionamiento de los conversores.

Además de todo esto, se ha utilizado el depurador de PyCharm para comprobar la persistencia de los datos al cambiar de formato. Gracias a este depurador, se ha podido inspeccionar a mano las variables de los eventos.

Por ejemplo, se puede observar fácilmente si el *timestamp* es correcto, ya que los eventos siempre se almacenan ordenados de menor a mayor por esta variable. También se puede comprobar si las coordenadas están mal interpretadas, ya que estas deben tener un valor entre 0 y el tamaño de la imagen. Si cualquier coordenada se halla fuera de este intervalo, se puede asegurar que ha ocurrido un error.

Usando la batería de funciones implementadas, los diferentes datasets y el depurador de PyCharm, se ha podido comprobar el correcto funcionamiento de todos los conversores.

6

Conclusiones y trabajo futuro

En este capítulo final, se mostrarán las conclusiones obtenidas tras el desarrollo del proyecto. Adicionalmente, se propondrán posibles mejoras a la aplicación para un trabajo futuro.

6.1. Conclusiones

El objetivo principal de este proyecto era desarrollar un conversor de archivos generados a partir de cámaras de eventos. Estos archivos se encuentran a lo largo de la red en diversos formatos, muchos incompatibles entre sí. Esta disparidad de formatos impide a muchos grupos de investigadores el uso de herramientas que soportan formatos diferentes.

Por ello, se decidió desarrollar un conversor de formatos, que permitiese cambiar la configuración de archivo rápidamente entre diversos tipos. El sistema presentado permite realizar esta conversión entre cinco formatos principales: bag, txt, mat, bin y AEDAT . Además, del formato mat se aceptan tres formatos diferentes, y del formato AEDAT se aceptan tres versiones diferentes. Estos formatos son los más utilizados a día de hoy en el campo de las cámaras de eventos, por eso se han escogido para ser implementados.

Por último, se han añadido a la aplicación diversas funcionalidades que mejoran la usabilidad del sistema. Se implementó diversas IU para que el usuario del sistema escoja el más apropiado para él. Además, se añadió un archivo de configuración, para poder automatizar las conversiones fácilmente.

Todo el programa está subido a la plataforma GitHub [1] para que cualquier investigador pueda utilizarlo gratuitamente. En este repositorio se ha añadido un archivo README para guiar al usuario.

6.2. Trabajo futuro

Tras la finalización del desarrollo del proyecto, se plantean diversas mejoras de cara a un trabajo futuro.

Para comenzar, se podrían aumentar el número de formatos compatibles. Por ejemplo, se podría añadir soporte al formato HDF5 . Este formato está empezando a aparecer en diversos datasets de cámaras de eventos y sería buena idea implementar su conversión correspondiente. El programa desarrollado permite la introducción de nuevos conversores gracias a la modularidad del código.

Otra mejora sustancial sería la implementación de conversores con información auxiliar. Es decir, algunos formatos admiten llevar entre los datos información complementaria a los eventos, como puede ser la calibración de la cámara o el IMU . Se podría desarrollar un sistema que, dependiendo de la configuración de archivo de entrada y de salida, añada estos parámetros a los datos convertidos.

La última mejora propuesta sería separar los archivos de configuraciones en lectura y escritura. Actualmente, el archivo de configuraciones se utiliza para ambas operaciones. Se plantea la posibilidad de separar esta funcionalidad en dos partes para permitir mayor maniobrabilidad a la hora de automatizar las conversiones.

Bibliografía

[1] A. C. Ayuso, "Event conversor." https://github.com/acalderon0/event_conversor, 2021.

[2] G. Gallego, T. Delbruck, G. Orchard, C. Bartolozzi, B. Taba, A. Censi, S. Leutenegger, A. J. Davison, J. Conradt, K. Daniilidis, and D. Scaramuzza, "Event-based vision: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Aug 2020. (Acceder).

[3] T. Delbruck, "The slow but steady rise of the event camera," *EE Times*, Apr 2020. (Acceder).

[4] M. Muglikar, M. Gehrig, D. Gehrig, and D. Scaramuzza, "How to calibrate your event camera," *Dept. Informatics, Univ. of Zurich and Dept. of Neuroinformatics, Univ. of Zurich and ETH Zurich*, 2021. (Acceder).

[5] Anónimo, "What is imu? inertial measurement unit working and applications," *Arrow*, Dec 2018. (Acceder).

[6] C. Rauch, "Esim: an open event camera simulator." https://github.com/uzh-rpg/rpg_esim, 2018.

[7] E. Mueggler, H. Rebecq, G. Gallego, T. Delbruck, and D. Scaramuzza, "The event-camera dataset and simulator: Event-based data for pose estimation, visual odometry, and slam," *International Journal of Robotics Research*, Feb 2017. (Acceder).

[8] Y. Bi, A. Chadha, A. Abbas, , E. Bourtsoulatze, and Y. Andreopoulos, "Graph-based object classification for neuromorphic vision sensing," in *2019 IEEE International Conference on Computer Vision (ICCV)*, IEEE, 2019.

[9] G. Gallego, "Event-based vision," *Sites Google*, Accedido 2021. (Acceder).

[10] A. Rodnitzky, "Event cameras: Where are they now?," *Medium*, Dec 2020. (Acceder).

[11] "Dynamic vision sensor (dvs)." https://www.youtube.com/watch?v=0ZEM57DZJes, Jul 2016.

[12] D. Tedaldi, G. Gallego, E. Mueggler, and D. Scaramuzza, "Feature detection and tracking with the dynamic and active pixel vision sensor davis." https://www.youtube.com/watch?v=nglfEkiK308, Feb 2017.

[13] S. Afshar, A. P. Nicholson, A. van Schaik, and G. Cohen, "Event-based object detection and tracking for space situational awareness.," *Cornell University*, Nov 2019. (Acceder).

[14] G. Orchard, G. Cohen, A. Jayawant, , and N. Thakor, "Converting static image datasets to spiking neuromorphic datasets using saccades," *Frontiers in Neuroscience*, vol. 9, Oct 2015. (Acceder).

[15] A. Vasudevan, P. Negri, B. Linares-Barranco, and T. Serrano-Gotarredona, "Introduction and analysis of an event-based sign language dataset," *IEEE International Conference on Automatic Face and Gesture Recognition*, 2020. (Acceder).

[16] A. Sironi, M. Brambilla, N. Bourdis, X. Lagorce, and R. Benosman, "Hats: Histograms of averaged time surfaces for robust event-based object classification," *Cornell University*, Mar 2018. (Acceder).

[17] Z. Wang, P. Duan, O. Cossairt, A. Katsaggelos, T. Huang, and B. Shi, "Joint filtering of intensity images and neuromorphic events for high-resolution noise-robust imaging," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2020.

[18] H. Rebecq, R. Ranftl, V. Koltun, and D. Scaramuzza, "High speed and high dynamic range video with an event camera," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019. (Acceder).

[19] M. Almatrafi, R. Baldwin, K. Aizawa, and K. Hirakawa, "Distance surface for event-based optical flow," *Cornell University*, Mar 2020. (Acceder).

[20] G. Gallego, T. Delbruck, G. Orchard, C. Bartolozzi, B. Taba, A. Censi, S. Leutenegger, A. Davison, J. Conradt, K. Daniilidis, and D. Scaramuzza, "Event-based vision resources." `https://github.com/uzh-rpg/event-based_vision_resources#survey_paper`, 2021.

[21] E. León, "Python: El lenguaje del futuro," *Baoss*, Dec 2020. (Acceder).

[22] "Introducción a json." `https://www.json.org/json-es.html`, Accedido 2021.

[23] "Pycharm." `https://www.jetbrains.com/es-es/pycharm/`, Accedido 2021.

[24] "Pylint." `https://pypi.org/project/pylint/`, Accedido 2021.

[25] "Ros noetic ninjemys." `http://wiki.ros.org/noetic`, Última revisión 2020-08-31.

[26] "Matlab." `https://es.mathworks.com/products/matlab.html`, Accedido 2021.

[27] "Bags format 2.0." `http://wiki.ros.org/Bags/Format/2.0`, Última revisión 2013-09-06.

[28] E. Mueggler, "Event.msg." `https://github.com/uzh-rpg/rpg_dvs_ros/blob/master/dvs_msgs/msg/Event.msg`, 2015.

[29] E. Mueggler, "Eventarray.msg." `https://github.com/uzh-rpg/rpg_dvs_ros/blob/master/dvs_msgs/msg/EventArray.msg`, 2015.

[30] "rosbag cookbook." `http://wiki.ros.org/rosbag/Cookbook`, Última revisión 2021-03-10.

[31] "Creating a ros msg and srv." `http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv`, Última revisión 2020-05-20.

[32] K. Conley, "Package rospy." `http://docs.ros.org/en/jade/api/rospy/html/rospy-module.html`, 2017.

[33] "Input and output (scipy.io)." `https://docs.scipy.org/doc/scipy/reference/io.html`, Última revisión 2021-04-26.

[34] Anónimo, "Binary file definition," *The Linux Information Project*, Feb 2006. (Acceder).

[35] E. Bahit, "Sobre el objeto file," *uniwebsidad*, Accedido 2021. (Acceder).

[36] "Aedat file formats." `https://inivation.github.io/inivation-docs/Software%20user%20guides/AEDAT_file_formats.html`, Accedido 2021.

[37] "Flatbuffers." `https://google.github.io/flatbuffers/`, Accedido 2021.

[38] L. Longinotti, "dv-runtime." `https://gitlab.com/inivation/dv/dv-runtime/-/`

tree/master, 2021.

[39] "Aedat." https://pypi.org/project/aedat/, Accedido 2021.

[40] V. Savikko, "Design patterns in python," *Software Systems Laboratory Tampere University of Technology*, Accedido 2021. (Acceder).

[41] "Singleton en python." https://refactoring.guru/es/design-patterns/singleton/python/example#example-0, Accedido 2021.

[42] S. Tobin-Hochstadt and E. Allen, "A core calculus of metaclasses," *Northeastern University and Sun Microsystems Laboratories*, Accedido 2021. (Acceder).

[43] "Python guis for humans." https://pysimplegui.readthedocs.io/en/latest/, Accedido 2021.

[44] "Codificador y decodificador json." https://docs.python.org/es/3.9/library/json.html, Última revisión 2021-06-14.

[45] "Parser for command-line options, arguments and sub-commands." https://docs.python.org/es/3.9/library/json.html, Última revisión 2021-06-14.

[46] L. Haoyi, "Build your own command line with ansi escape codes," *Haoyi's Programming Blog*, Jul 2016. (Acceder).

[47] "Catkin conceptual overview." http://wiki.ros.org/catkin/conceptual_overview, Última revisión 2020-03-26.

[48] R. Dhanawade, "Python traceback," *GeeksForGeeks*, Última revisión 2020-07-23. (Acceder).

Definiciones

big-endian Formato en el que se almacenan los bytes en orden "natural", es decir, el byte menos significativo se almacena en primer lugar.

catkin Es el sistema oficial para la construcción de ROS , sucesor del sistema original rosbuild [47].

dataset Colección de datos.

header Cabecera de un dato. Contiene información adiccional acerca del dato que está acompañando, como puede ser la fecha de creación o la última fecha de modificación.

little-endian Formato en el que se almacena el byte menos significativo en último lugar. Se considera el orden "no natural" de almacenaje de datos.

parseo Proceso de analizar una secuencia de símbolos para extraer datos de ella, también conocido como análisis de sintaxis.

path Cadena de texto que representa la ruta hacia un archivo o carpeta de un ordenador.

serializar Proceso por el cual se codifica un objeto para que sea más compacto.

timestamp Secuencia de caracteres que denominan la hora y la fecha en el que sucedió un evento determinado. También se le conoce como "marca temporal".

tópico En los archivos bag, es la etiqueta que define a un mensaje, para luego poder filtrar los mensajes por estos tópicos.

traceback Reporte informativo con la lista de llamadas a funciones hasta un punto del programa determinado [48]. Se suele usar para analizar y reparar errores.

Acrónimos

AEDAT *Address Event Data*.

ANSI *American National Standards Institute*.

APS *Active Pixel Sensor*.

ASCII *American Standard Code for Information Interchange*.

DAVIS *Dynamic and Active-pixel Vision Sensor*.

DVS *Dynamic Vision Sensor*.

HDF5 *Hierarchical Data Format 5*.

IDE *Integrated Development Environment*.

IMU *Inertial Measurement Unit*.

IU Interfaz de Usuario.

JSON *JavaScript Object Notation*.

MATLAB *Matrix Laboratory*.

ROS *Robot Operating System*.

UZH Universidad de Zúrich.

Apéndices

A

Archivo README

En este apéndice se mostrará el contenido del archivo **README** de la aplicación. Este archivo contiene toda la información que el usuario necesita para poder ejecutar la aplicación. En este apéndice se muestra el contenido del archivo sin seguir el mismo formato para seguir un mismo estilo en todo el documento.

Al principio del **README** se presenta una breve introducción para el usuario que acceda al repositorio y quiera informarse rápidamente del contenido de este:

En este proyecto se ha desarrollado un conversor de archivos generados por cámaras de eventos. Esta aplicación soporta los ficheros de formato bag, mat, txt, bin y AEDAT y está programada en Python3.

A.1. Introducción al programa

Este programa se centra en la conversión de archivos que almacenan *eventos*. Estos *eventos* son producidos por las cámaras de eventos y se almacenan con diferentes formatos. El programa desarrollado puede leer los eventos almacenados en un archivo y escribirlos en otro archivo con diferente formato. Este programa solo transforma los eventos almacenados, el resto de la información que pueda contener el archivo (la calibración de la cámara por ejemplo) se perderá en la conversión.

A.2. Pre-requisitos

Se debe tener instalado Python3. Se ha programado el código con la versión 3.9.2, por lo que se recomienda utilizar esta versión por posibles errores.

También se debe tener instalado una versión de ROS para poder acceder a las cabeceras compatibles con los archivos bag. Se recomienda instalar la versión Noetic ya que es la usada en el desarrollo. En este link se ofrece un tutorial para su instalación.

A.3. Dependencias

En esta aplicación se han utilizado las siguientes librerías de Python:

- aedat
- genpy
- NumPy
- PySimpleGUI
- rosbag
- rospy

A.4. Instalación

Se deben instalar todas las dependencias antes de ejecutar el programa. Para instalar estos paquetes, se debe ejecutar el siguiente comando:

```
pip3 install aedat genpy numpy pysimplegui rosbag rospy
```

Cuadro A.1: Comando instalación dependencias

A.5. Ejecución y parámetros

Para iniciar la aplicación, se debe ejecutar el archivo **main.py** del repositorio. Se debe ejecutar con python y tiene dos tipos de argumentos:

- Posicionales, deben seguir un orden específico, hay dos:
  - **input_file**: Debe ir en primer lugar, path del fichero de entrada.
  - **output_file**: Debe ir en segundo lugar, path del fichero de salida.
- No posicionales, no deben seguir un orden, hay seis:
  - **- -input_type**: Debe acompañarlo una cadena de texto indicando el tipo de entrada.
  - **- -output_type**: Debe acompañarlo una cadena de texto indicando el tipo de salida.
  - **- -config_path**: Debe acompañarlo una cadena de texto que indique el path al fichero de configuración.
  - **- -use_config**: El uso de este flag indica al programa la utilización del archivo de configuración.
  - **- -use_graphic_UI**: El uso de este flag indica la utilización de la interfaz gráfica.
  - **- -use_terminal_UI**: El uso de este flag indica la utilización de la interfaz basada en la terminal.

Si no se usa ni la flag **- -use_graphic_UI** ni la flag **- -use_terminal_UI**, no se mostrará nada de información al usuario. Además, en este modo se deben usar los argumentos **input_file**, **- -output_type**

y **- -use_config**.

Si se usa la opción **- -use_terminal_UI**, se deberá introducir los parámetros **input_file** y **- -output_type**.

Introducir el comando **- -help** para mostrar ayuda adicional de los argumentos.

A continuación, se muestra un ejemplo de ejecución utilizando la interfaz gráfica, con fichero de entrada "*input.bag*", tipo de salida "*bag*", path del fichero de configuración "*config.json*ż usando el fichero de configuración.

```
python3 main.py input.bag - -output_type bag - -use_config - -config_path
config.json - -use_graphic_UI
```

Cuadro A.2: Ejemplo de ejecución del programa

A.6. Archivo de configuración

En este programa se añade un fichero de configuración para automatizar las conversiones. Algunos formatos requieren datos extra, por ejemplo, el tipo *mat* requiere el nombre de las variables a almacenar. Para automatizar este proceso, se ha implementado un fichero de configuración que guarda estos valores.

El fichero usado por la aplicación por defecto se encuentra en la ruta *src/config/config.json*. En este fichero se puede observar el formato que debe tener un fichero de configuración, así como cada variable explicada para que el usuario pueda implementar un fichero de configuración por sí mismo.

A.7. Formatos aceptados

En esta aplicación se aceptan cinco formatos diferentes:

- Archivos *bag*, utilizados por ROS. El formato es el mencionado en la siguiente web.
- Archivos *txt*, que contienen texto plano. El formato es el mencionado en la siguiente web.
- Archivos *mat*, utilizados por MATLAB. Se aceptan tres estructuras diferentes:
  - Se almacenan cuatro variables dentro de una estructura. Cada variable contiene un array de datos.
  - Se almacenan cuatro variables directamente. Cada variable contiene un array de datos.
  - Se utiliza para almacenar una matriz 4 x N, siendo N el número de eventos.
- Archivos *bin*, que contienen los datos codificados en binario.
- Archivos *AEDAT*, que siguen el formato especificado en esta web. Se aceptan las siguientes versiones:
  - Versión **2.0**, escritura y lectura implementadas.

- Versión **3.1**, escritura y lectura implementadas.
- Versión **4.0**, solo lectura implementada.

A.8. Autoría

Este programa se ha desarrollado como Trabajo de Fin de Grado de la Escuela Politécnica Superior (EPS) de la Universidad Autónoma de Madrid (UAM) en el grado de Ingeniería Informática.

Autor: Andrés Calderón Ayuso
Tutor: Erik Velasco Salido
Ponente: José María Martínez Sánchez

Cuadro A.3: Autores del proyecto